Efficient Parallel Graph Coloring with Prioritization

Laxmikant V. Kale, Ben H. Richards and Terry D. Allen

Department of Computer Science

University of Illinois

Urbana, IL 61801

kale@cs.uiuc.edu

http://charm.cs.uiuc.edu

Objectives

To color a graph with C colors, such that no two adjacent nodes have the same colors.

Relevance

- Proven to be NP complete, but there are some good heuristics.
- Good search problem - techniques developed here can be applied to other search problems.

Aims

- To get the best sequential as well as parallel performance
  - Not just speedups
- Find a coloring, or determine if there are no C colorings.
- Develop techniques applicable to other search problems as well.

Search progression

```
Read Graph and build initial state

On each processor:
   if (unevaluated states exist)
       choose a state to evaluate.
       if (uncolored nodes exist)
          choose an uncolored node to expand on.
          for each color available to that node:
             assign the color, and create a new
             state.
       else
          report success.
   else
      report failure
```

Heuristics

"It is difficult to get good speedups using good heuristics"
— they make the search more irregular.

- Variable ordering
- Value ordering
- Precoloring
- Node removal

Variable ordering

Choosing the uncolored node in a state to assign colors to.

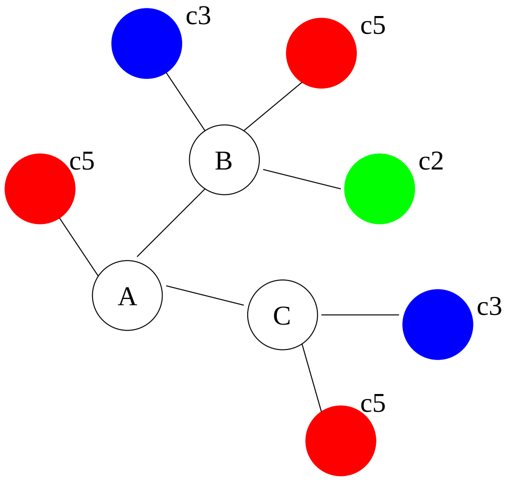
- choose the one with smallest number of colors available to it.
- handle the most difficult cases first (decreasing the amount of searching to be done later)

Value ordering

Prioritize the coloring schemes

- Search subtrees more likely to contain a solution first.
- Less constraining choices.
- Using bitvector priorities

Prioritization



Color for A	Neighbors Affected	Heuristic Value	Rank	Bit-Vector Priority
c1	B and C	3	2	10
c2	C	4	1	01
c3	Neither.	5	0	00
c4	B and C	3	3	11

Redundancy in color assignments

For each color assignment, switching the colors of all nodes colored C1, and C2 produces an equivalent but different coloring. With C colors, the redundancy is C!.

Precoloring

Fix the colors for some of the nodes without loss of generality

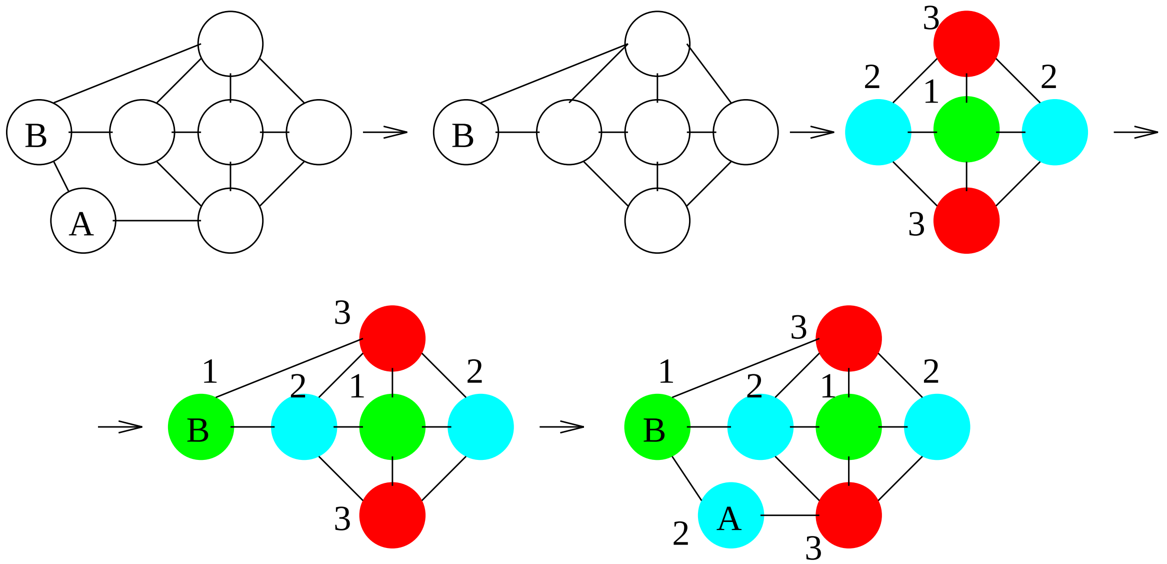
- Bring redundancy down from C! to (C-2)! or even (C-3)!
- Failed searches have fewer states to search.

Node Removal

Remove the nodes in the graph which can be colored no matter what colors are assigned to neighbours (more colors available to it than uncolored neighbours)

- Reduces number of states
- Done recursively - removing a node may allow some of its neighbours to also be removed.

Node removal example: 3-coloring.



Refinements

- Impossibility testing
- Forced moves
- Split Graphs

Impossibility testing

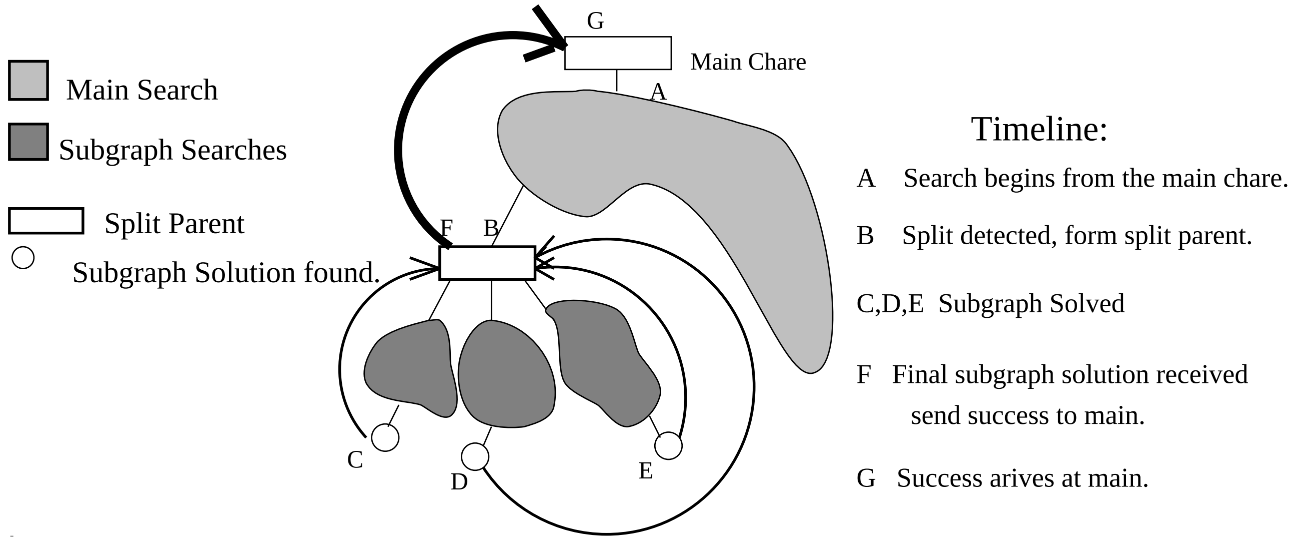
Do not create states which have nodes with zero available colors.

Forced moves

If the state to be created has a node with one available color, skip it and go directly to the states it will create.

Split graphs

- Work on coloring the different parts of an unconnected subgraph in parallel.
- All of the subparts must be colorable for a solution to be reported.



Parallel Refinements

Kill Chasing

- With split graphs, failure of one branch, can be used to stop the work on the other.
- Used to stop the computations once a solution is found.

Grainsize Control

- Ensure a minimum average grainsize per chare.
- Process states in sequence using a stack, until enough work has been done to merit the creation of child chares.

Performance Results

- Definite performance gains due to addition of heuristics
  - Value ordering helps with colorable graphs
  - Variable ordering, Precoloring and Node removal helps with all graphs

Performance results

File	Nodes	Edges	Colors	Solution
Example 4	300	1626	5	Yes
Example 7	450	2451	5	Yes
Example 8	600	2338	3	No
Example 9	301	4274	5	No

Table 1: Input file summary.

Processors	Chares	Execution Time (sec)	Time per Chare (ms)
1	1	1440	1440177
2	1463	602	823
4	2138	311	582
8	2834	199	562

Table 2: Parallel speed-ups on a successful search on Multimax.

GS setting	540	570	580	585
Processors	Execution Time (ms)			
16	100907	94093	99954	193217
32	52187	46182	62686	145633
64	28182	26500	47837	129824
128	15648	16147	30022	96426
Statistic	Statistic Value (across processors)			
mean GS	30	59	436	2980
std. dev	11	82	991	6599
min-max GS	9-93	14-2411	17-8342	17-39689
Chares	50552	24075	2997	434

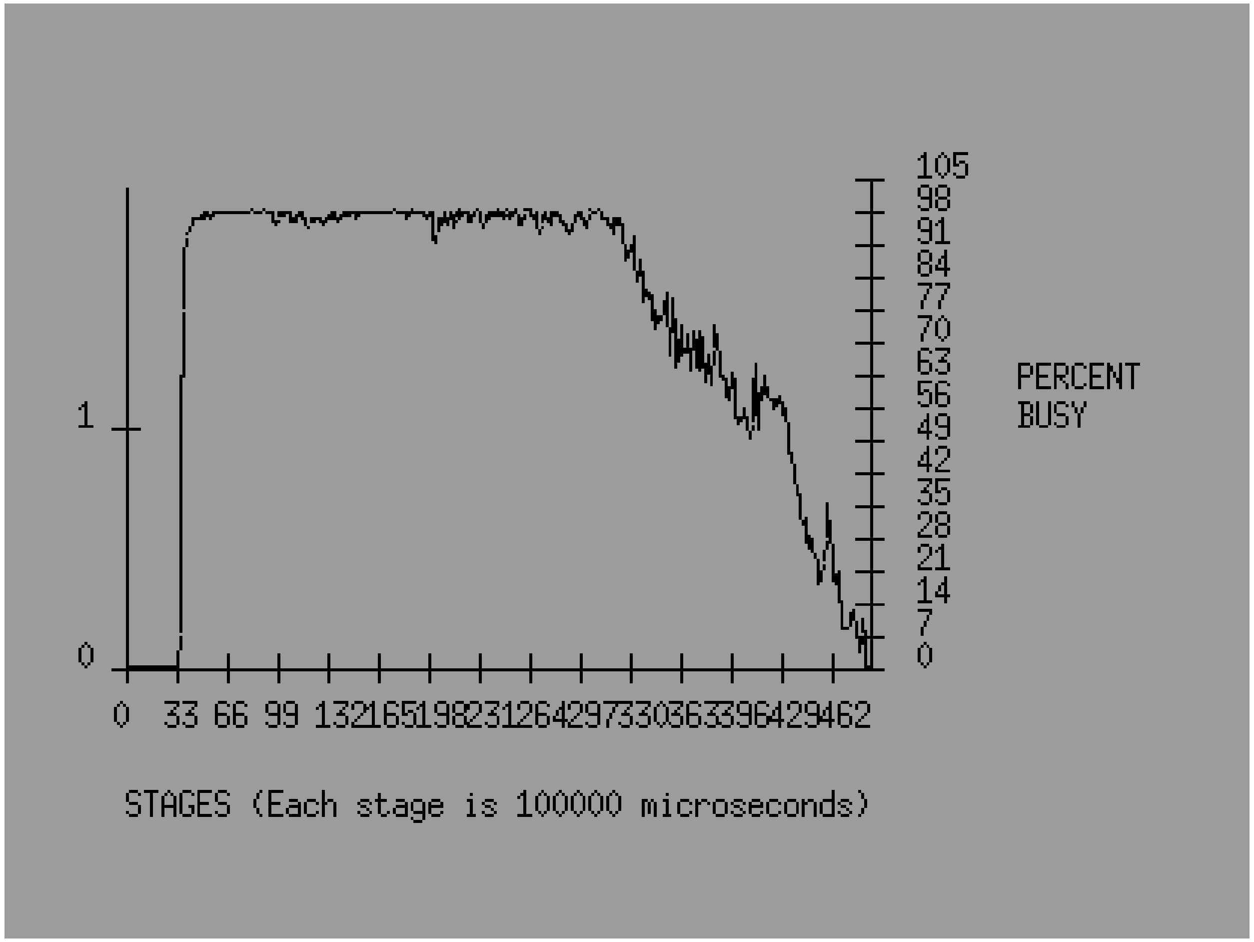
Table 3: Example 8 results on nCUBE/2.

Stack based grainsize control

- Somehow, we must avoid large grains, while keeping the average grainsize at a reasonable value.
- Estimating work under a node is hard.
- Idea: let a process procreate only when it has worked "Enough".
- Each chare (process) maintains a stack, and fires k children each time it crosses a threshold of work completed.
- Children development are fired from the bottom of the stack.
- This, or similar, method has been used by Halstead et al.

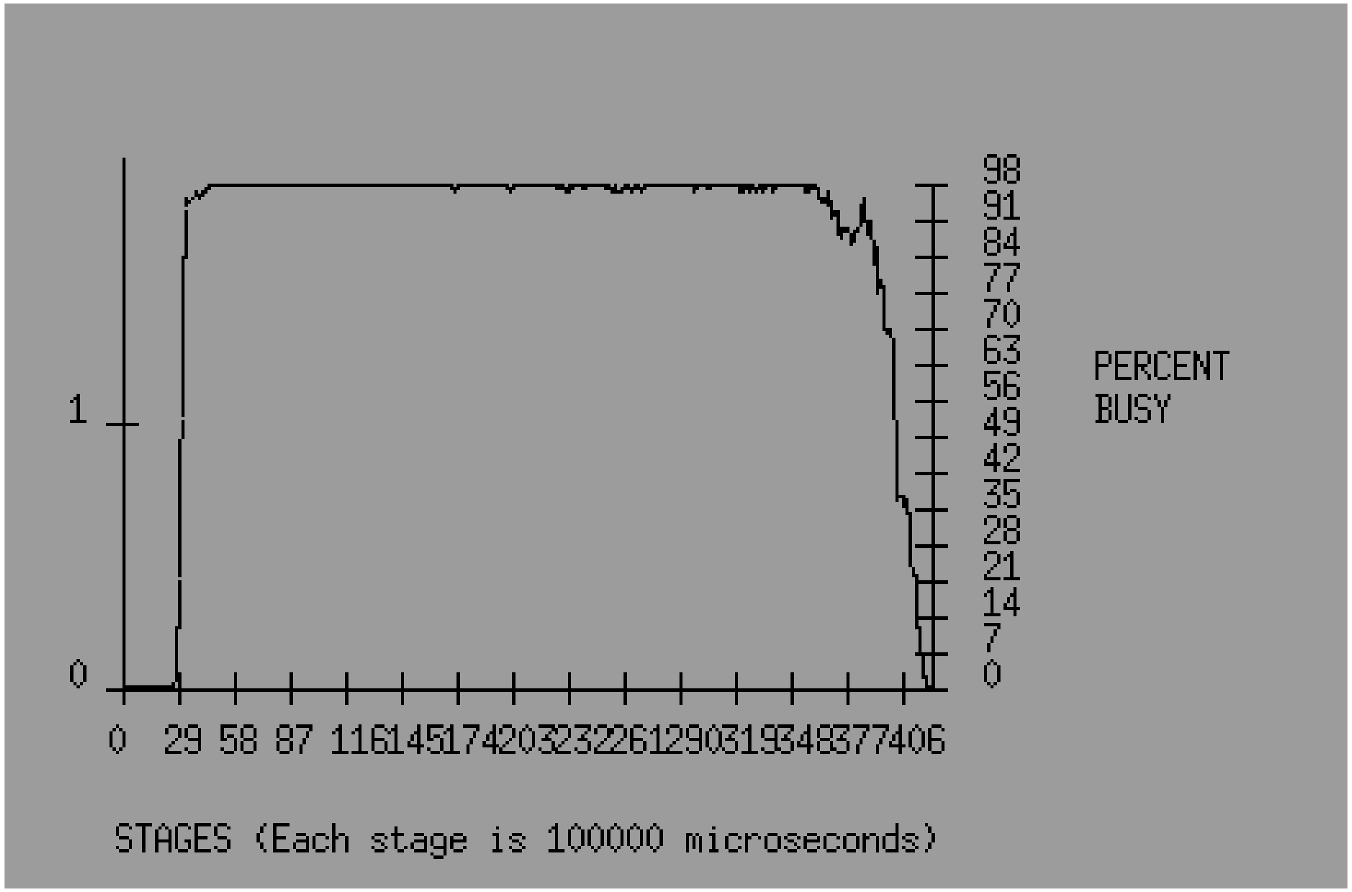
Utilization Plot

With normal grainsize control:

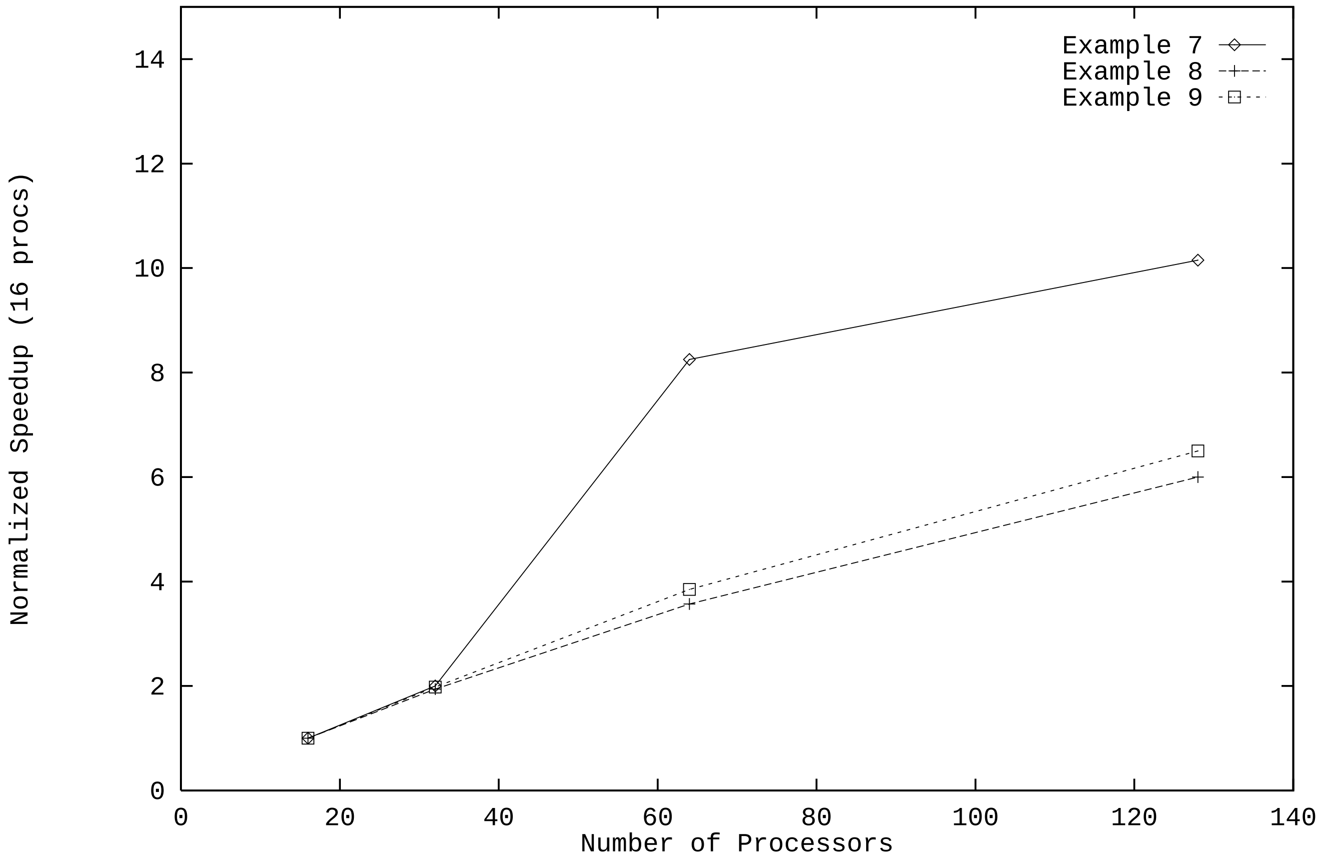


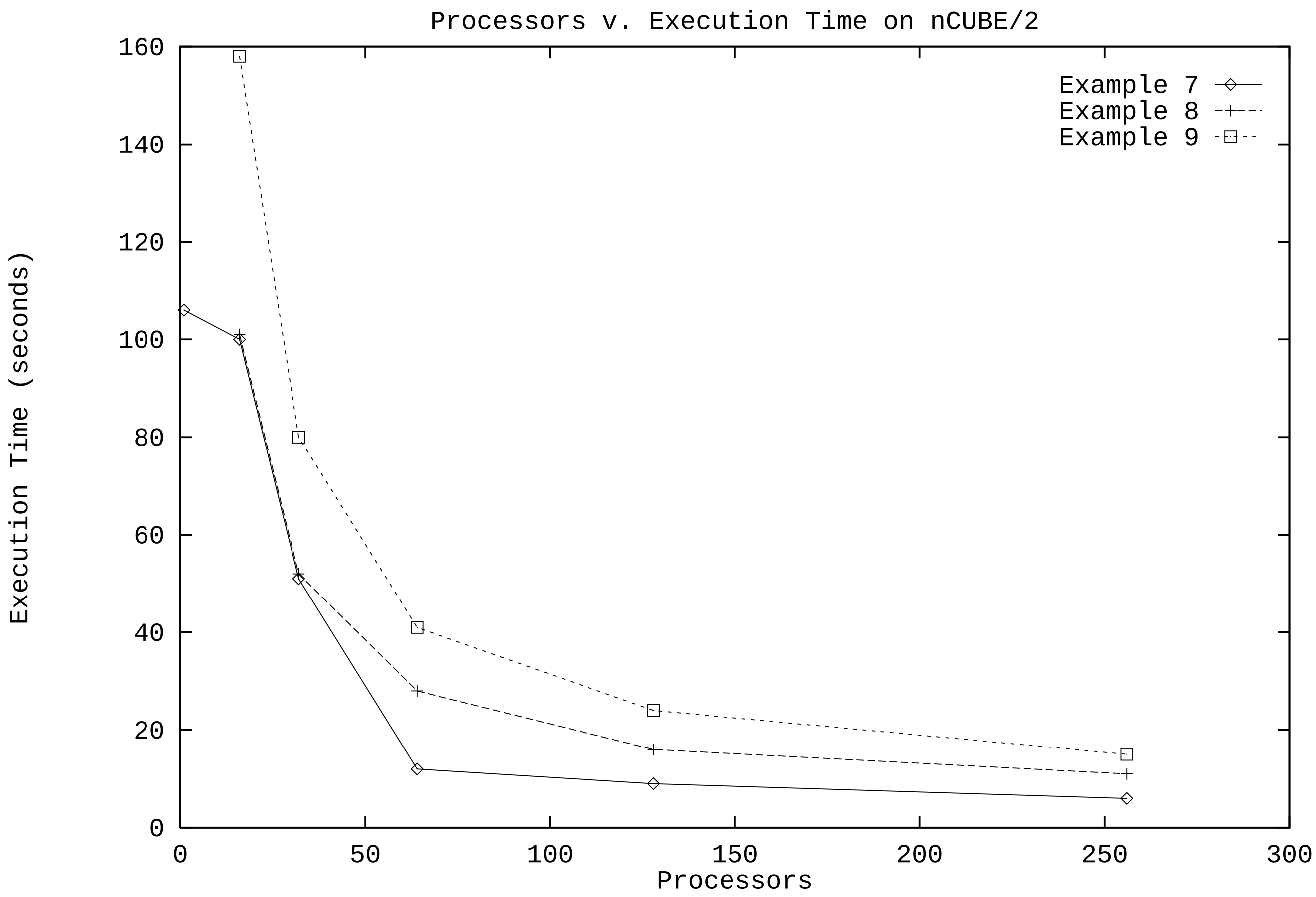
Utilization Plot

With new grainsize control:



More performance data




Processors v. Execution Time on nCUBE/2
Execution Time (seconds)
Processors
0
20
40
60
80
100
120
140
160
0
50
100
150
200
250
300
Example 7
Example 8
Example 9